Accessing the Intel® Random Number Generator with CDSA

Barry Pivitt
Intel Platform Security Division

Contents

INTRODUCTION 1

Scope of this Document 1

INTEL RANDOM NUMBER GENERATOR OVERVIEW 2

COMMON DATA SECURITY ARCHITECTURE OVERVIEW 2

ACCESSING INTEL PLATFORM SECURITY FEATURES THROUGH CDSA 4

Identifying CSP Support for the Intel RNG 4

Generating Random Numbers 6

Testing the Intel RNG 8

CONCLUSION 9

REFERENCES 9

Introduction

Intel Corporation’s vision for computing is "a billion connected PCs, a million connected servers." Security in computing is a fundamental requirement to make this happen. Consequently, Intel believes the industry should work together to develop the “Trusted, Connected PC”. The Trusted, Connected PC will enhance today’s PCs, servers, and workstations with security capabilities that allow users to perform secure transactions and information exchange over the Internet with privacy, and without fear of theft, viruses, or other attacks.

Developing a hardware foundation for security is one key element to making Intel’s security vision a reality. By implementing key security features within hardware for every computing platform, enhanced security will become ubiquitous. A security solution implemented in hardware is often more robust than a software solution, since hardware has the unique ability to hide execution and storage of information. Better security solutions implemented across all systems will pave the way towards increased connectivity, access to new products and services, and new business models.

In addition to ubiquitous hardware-based security features, Intel developed the Common Data Security Architecture (CDSA) to address application requirements. Today, an application implementing cryptographic or credential-based security features must select from various “point” solutions. These point solutions suffer from various problems: lack of support for specific platforms, inability to share credentials among applications, and limitations in cryptographic support. CDSA is an open, interoperable, and extensible security framework that provides basic security services for applications.

The combination of hardware-based security services and CDSA provides a compelling set of hardware and software security features for applications. These security features provide the necessary building blocks for robust applications in areas such as secure communications, electronic commerce, and content protection.

Scope of this Document

This paper details how applications can access Intel’s platform silicon security features through CDSA. Included is a brief overview of the first Intel platform silicon feature, the Intel Random Number Generator (RNG), as well as an introduction to the CDSA security framework. The paper details how independent software vendors (ISVs) can access the Intel RNG security features through the CDSA architecture, providing ISVs with an easy migration path from existing software-based solutions.

Intel Random Number Generator Overview

The Intel Random Number Generator is a fundamental building block for improving the ability of computers to preserve the confidentiality of electronic communications. Random number generation is a key component of the encryption algorithms that protect data. Most random number generators available on systems today are software RNGs (often called pseudo RNGs, or PRNGs), which are not capable of generating truly random data. Because pseudo RNGs generate random data by means of a fixed algorithm, their output can be predictable. This predictability weakens software-only encryption schemes relative to hardware-based systems.

Intel's silicon-based random number generator generates true[1] random numbers (numbers which are unpredictable and non-repeating), which can increase the strength of an encryption system. Intel's random number generator passes all FIPS 140-1[2] RNG statistical tests, making it a preferred solution wherever random numbers are required. The Intel hardware random number generator is a key component for use in any strong security solution.

Businesses and consumers rely on networks for communication, using the PC as their protected entry point. In a world that increasingly depends on digital information, PC users can now justifiably expect more security. Intel's silicon-based RNG provides a strong foundation for computer data security. ISVs demand a ubiquitous platform upon which to deploy their enhanced solutions. Intel's hardware-based security building blocks enable OEMs to deliver new security technology on broadly deployed IA (Intel Architecture) platforms and to enhance their product lines with a new category of security-enabled systems.

Common Data Security Architecture Overview

CDSA provides a set of software-based security building blocks for ISVs that complement the Intel RNG. CDSA is designed as an overall infrastructure for data security on PCs, workstations, and servers. It is founded on two fundamental data security premises: *digital certificates* (a form of electronic identification that enables a hierarchy of trust, dependent on the identity of the user) and *portable digital tokens*, which store cryptographic keys and perform cryptographic operations.

CDSA truly lives up to the term "infrastructure." It defines three layers, each building on the more fundamental services of the layer below it. Figure 1 shows the architecture in block-diagram form. The bottom layer is made up of service provider modules that start with basic components – cryptographic algorithms, base certificate manipulation facilities, and storage – and build up to secure, digital certificate-based transaction protocols in the uppermost layer (System Security Services). CDSA supports diverse programming environments, ranging from ANSI C to Java. The architecture is designed to be both modular and extensible. Extensibility is important because it encourages ISVs to develop incremental functionality and performance improvements to remain competitive.

---

[1] Some might argue that it is not possible to generate a "true" random number. This paper assumes Schneier's definition of true (or real) random number generators—they generate sequences that look random, are unpredictable, and cannot be reliably reproduced [4]. More detailed discussions of "true" versus "pseudo-" random numbers are presented in [1], [2], and [3].

[2] FIPS is the United States government's Federal Information Processing Standard. A publication of the National Institute of Standards and Technology, the FIPS 140-1 specification describes government requirements for cryptographic modules for sensitive, but unclassified use. For general information about FIPS 140, see the FAQ from Corsec Security, Inc. at http://www.corsec.com/FIPS140-1FAQ.html, or the FIPS 140-1 publication [5].

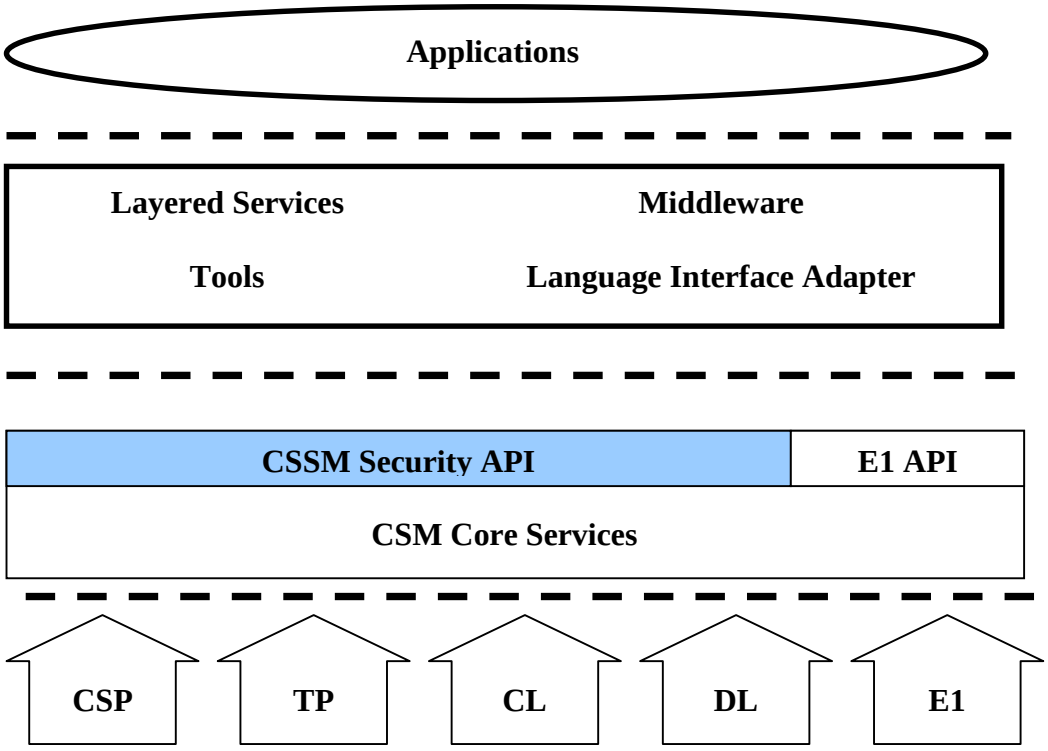

Figure 1: The CDSA Architecture

The heart of CDSA is the Common Security Services Manager (CSSM). CSSM is an API with openly published specifications[3] that an application can use to gain access to security features. Typical requested security features include performing cryptographic operations, determining the trust level of a certificate holder, manipulating certificates, and accessing data storage media.

CSSM gains specific security features from service provider modules. A service provider module provides a specific subset of security functions. Four types of standard service provider modules integrate into the CDSA environment. These are:

Cryptographic Service Provider (CSP) modules
CSPs perform cryptographic operations such as bulk encrypting, digesting, and digital signatures. In addition, they store private keys. CSPs are the "lock and key" components of the CDSA structure.

Trust Policy (TP) modules
TPs implement policies defined by authorities and institutions and set the level of trust required to carry out specific actions (such as issuing a check or access to confidential intellectual property). The modular concept permits TP modules to be associated with the needs of specific institutions. For example, a credit card issuer might have different trust policies than a government agency.

Certificate Library (CL) modules
CLs provide syntactic manipulation of stored certificates and revocation lists, as well as access to remote signing capabilities known as Certification Authorities (CA).

[3] The CDSA 2.0 specification was adopted by The Open Group* in December, 1997. Versions of the specification can be ordered from The Open Group, or downloaded from their website [6].

Data Storage Library (DL) modules
DLs provide stable storage for security-related data objects – certificates, cryptographic keys, policy objects and more. The actual storage may be in a commercially available database system, a native file system, a custom hardware device, and so on. DLs are analogous to a "file cabinet" for security data.

In addition to the four standard service provider modules, the CDSA 2.0 specification allows for elective module managers. These elective module managers provide the mechanism for adding new and compelling security features as the marketplace evolves.

Any number of independent software or hardware vendors can create service provider modules, highlighting the CDSA emphasis on openness and interoperability. Applications directly or indirectly select the service provider modules needed for specific security services.

Accessing Intel Platform Security Features through CDSA

Figure 2 shows the basic flow of an application request for use of Intel RNG features through CDSA. Applications make a call to one of the CSSM API functions, with parameters indicating the application's desire to use the Intel RNG. CSSM performs some basic processing, and then passes the call to the service provider module (in this case, a CSP) specified by the application. Based on the function call parameters, the CSP generates a random number using the Intel RNG, and passes the results back up the software stack.

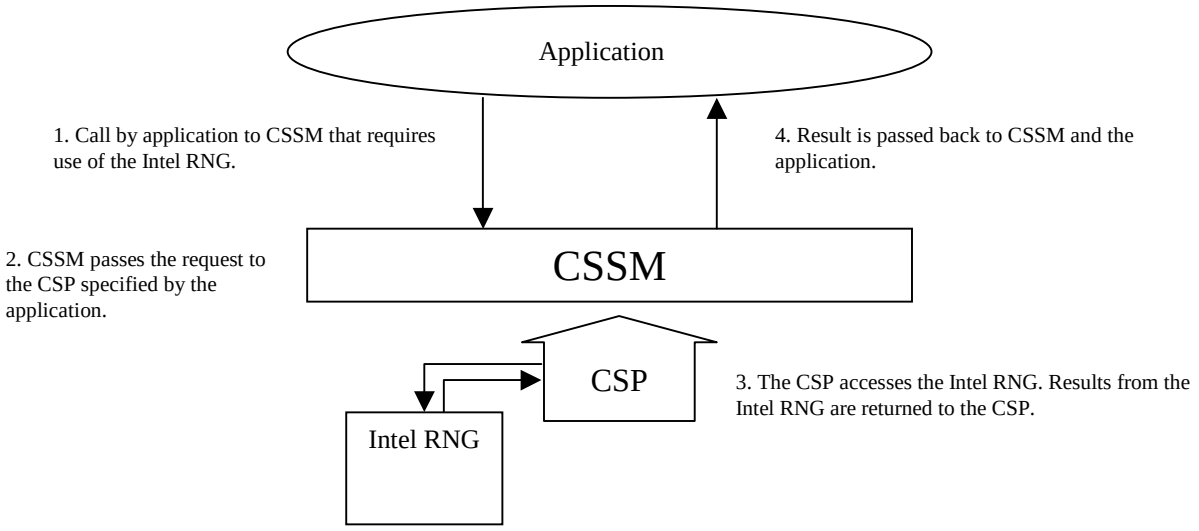

Figure 2: Control Flow for Intel RNG Requests using CDSA

The CDSA 2.0 specification includes CSSM APIs for random number generation. Applications can identify CSPs that support the Intel RNG using the **`MDSUTIL_ListModules`** and **`MDSUTIL_GetModuleInfo`** functions. Once an application identifies a CSP with Intel RNG support capabilities, the **`CSSM_CSP_CreateRandomGenContext`** and **`CSSM_GenerateRandom`** functions are used to generate random numbers. In addition, an application can test the Intel RNG using a CSSM passthrough function, **`CSSM_CSP_PassThrough`**.

Identifying CSP Support for the Intel RNG

When a CSSM service provider module is installed on a system, the module stores information about its capabilities in the Module Directory Services (MDS) database. The CDSA 2.0 specifications provide a definition of MDS, as well as the APIs used to access MDS. Applications use the MDS APIs to locate this information and identify service provider modules that support the requested function.

Applications that wish to use the Intel RNG must identify an appropriate CSP. Using the information stored in MDS, an application can quickly identify a CSP with Intel RNG support. Example 1 shows a code fragment of how to search the module information stored within MDS for CSPs with Intel RNG support:

```
CSSM_LIST_PTR pGUIDList = NULL;
CSSM_BOOL IntelSecurityDriverSupport = CSSM_FALSE;
CSSM_GUID IntelRngCsp;
CSSM_CSP_CAPABILITY_PTR pCapabilityList = NULL;
CSSM_MODULE_INFO_PTR pInfo = NULL;
CSSM_SERVICE_INFO_PTR pServiceInfo = NULL;
CSSM_CSPSUBSERVICE_PTR pSubService = NULL;
CSSM_CSP_CAPABILITY_PTR pCapability = NULL;
uint32 iModule = 0;
uint32 iService = 0;
uint32 iSubService = 0;
uint32 numCapabilities = 0;
uint32 iCapabilities = 0;
   // get a list of CSPs installed on the system
pGUIDList = MDSUTIL_ListModules(CSSM_SERVICE_CSP, CSSM_FALSE);

if (!pGUIDList)
{
  // Search through the CSP list for a CSP that supports the Intel RNG
  for(iModule = 0; iModule < pGUIDList->NumberItems; iModule++)
  {
    // Get the module info, which indicates whether the CSP supports the Intel RNG
    pInfo = MDSUTIL_GetModuleInfo(&(pGUIDList->Items[iModule].SubserviceUid.Guid),
                                  CSSM_SERVICE_CSP, CSSM_ALL_SUBSERVICES, CSSM_INFO_LEVEL_ALL_ATTR,
                                  CSSM_USEE_NONE);

    // Search each within each service of the module
    for(iService = 0; iService < pInfo->NumberOfServices; iService++)
    {
      pServiceInfo = &pInfo->ServiceList[iService];

      // search each subservice contained in the service of the module
      for(iSubService = 0; iSubService < pServiceInfo->NumberOfSubServices; iSubService++)
      {
        pSubService = (CSSM_CSPSUBSERVICE_PTR)(&pServiceInfo->SubServiceList[iSubService]);

        // If the subservice is for a CSSM_CSP_HYBRID type, get the capabilities
        if(pSubService->CspType == CSSM_CSP_HYBRID)
        {
          for(iCapabilities = 0;
              iCapabilities < pSubService->SubServiceInfo.HybridCspSubService.NumberOfCapabilities;
              iCapabilities++ )
          {
            pCapability = &pCapabilityList[iCapabilities];
```

```
            // If the CSSM_ALGID_IntelPlatformRandom algid capability exists, this
            // CSP supports the Intel RNG
            if( pCapability->AlgorithmType == CSSM_ALGID_IntelPlatformRandom )
            {
              // CSP supports the Intel RNG. At this point, you can set variables
              // break out of the search loop, etc.
              IntelSecurityDriverSupport = CSSM_TRUE;
              IntelRngCsp = pGUIDList->Items[iModule].SubserviceUid.Guid;
            }
          }
        }
      }
    }
  }
}
```

Example 1: Searching CSPs for Intel RNG Support

While this example may seem lengthy, the processing is straightforward. An application obtains a list of CSPs installed on the system through the **`MDSUTIL_ListModules`** function. Once the application obtains a list of installed CSPs, the application calls **`MDSUTIL_GetModuleInfo`** to get the CSP's module information. The application traverses the CSP module information through the service, subservice, and capability information. The CSP supports the Intel RNG feature if the application finds a CSP capability containing the `CSSM_ALGID_IntelPlatformRandom` algorithm ID.

Once an application identifies a CSP with Intel RNG support, the application can attach the CSP module with the **`CSSM_ModuleAttach`** function.

Generating Random Numbers

To generate random numbers, applications use two CSSM APIs: **`CSSM_CSP_CreateRandomGenContext`**, and **`CSSM_GenerateRandom`**. Example 2 shows how applications typically create random numbers using a software random number generator (pseudo RNG):

```
// Example of how to generate a random number via CSSM, using a
// cryptographic algorithm

#define RANDOM_NUMBER_SIZE 16 //Will generate 16 bytes of random data

CSSM_CSP_HANDLE hCSP;
CSSM_CC_HANDLE hCC;
CSSM_DATA randomNumber;
CSSM_RETURN retval;
uint8 buffer[RANDOM_NUMBER_SIZE];

// Assume CSP that supports MD2 RN generation, hCSP, was already found and attached
// For the seed value pass NULL. The CSP will use its default seed.
hCC = CSSM_CSP_CreateRandomGenContext(hCSP, CSSM_ALGID_MD2Random, NULL,
                                      RANDOM_NUMBER_SIZE);

// Create random number using context generated above
if ( hCC != NULL )
{
  randomNumber.Length = RANDOM_NUMBER_SIZE;
  randomNumber.Data = buffer;
  memset(randomNumber.Data, 0, RANDOM_NUMBER_SIZE);
  retval = CSSM_GenerateRandom(hCC, &randomNumber);

  // randomNumber can now be used as random data
}
```

Example 2: Typical Random Number Generation via CSSM API Calls

An application calls **CSSM_CSP_CreateRandomGenContext** to create a *security context*. A security context is a run-time structure containing security-related execution parameters and some internal CSSM state information. After security context generation, the application passes the context into the **CSSM_GenerateRandom** function. This function uses the security context information to generate the random number.

When the security context is created, an application passes in two important parameters. One important security context parameter is the random number generation method. In the above example, the security context identifies the MD2 algorithm as the method for random number generation. For software-based CSPs, this indicates the use of a pseudo RNG.

The second important parameter during security context creation is the seed value. For pseudo RNGs, the seed value is critical to ensure the random number is not predictable. An outside entity that knows the seed value used can quickly predict the value of the random number generated by a pseudo RNG. In the above example, the seed value is NULL, which indicates that the CSP will create a default seed value. The methodology for generating the seed value is unknown to the application developer.

To use the Intel RNG, an application developer passes a unique algorithm ID during security context creation. Example 3 shows how an application generates random numbers using the hardware-based Intel RNG.

```
// Example of how to generate a random number via CSSM using the
// Intel RNG

#define RANDOM_NUMBER_SIZE 16      //Generate 16 bytes of random data

CSSM_CSP_HANDLE hCSP;
CSSM_CC_HANDLE hCC;
CSSM_DATA randomNumber;
CSSM_RETURN retval;
uint8 buffer[RANDOM_NUMBER_SIZE];

// Assume CSP that supports MD2 RN generation, hCSP, was already found and attached
// NOTE: the seed value is NULL, but it is NOT required with the Intel hardware RNG!
hCC = CSSM_CSP_CreateRandomGenContext(hCSP, CSSM_ALGID_IntelPlatformRandom, NULL,
                                      RANDOM_NUMBER_SIZE);

// Create random number using context generated above
if ( hCC != NULL )
{
  randomNumber.Length = RANDOM_NUMBER_SIZE;
  randomNumber.Data = buffer;
  memset(randomNumber.Data, 0, RANDOM_NUMBER_SIZE);
  retval = CSSM_GenerateRandom(hCC, &randomNumber);

  // randomNumber can now be used as random data
}
```

Example 3: Random Number Generation using the Intel RNG via CSSM API Calls

In order to use the Intel RNG, an application generates a security context using the algorithm ID `CSSM_ALGID_IntelPlatformRandom`. Instead of using a pseudo RNG technique, the CSP obtains a random number from the Intel RNG. The details associated with accessing the Intel RNG are transparent to the application requesting a random number.

An important point between the pseudo RNG example (Example 2) and the Intel RNG example (Example 3) is the amount of code differences. The only difference between the two examples is the algorithm ID used to create a security context. Because the application calls the CDSA framework for security services, a single, consistent API is used to switch between the two RNG methodologies.

What about the seed value? In Example 3, creation of the security context still uses a NULL seed value. However, since the Intel RNG calculates a random number based on the physical environment and not a software algorithm, a seed value is not required.

Testing the Intel RNG

Depending on the application, an ISV may want to evaluate the quality of the random numbers generated by the Intel RNG at key points within the application's execution. While determining the "randomness" of a random number is not an easy task, an application can perform the FIPS 140-1 statistical RNG tests on the Intel RNG. The FIPS 140-1 publication specifies four statistical tests performed on 20,000 bits of random data. By performing the FIPS 140-1 tests, an ISV has an indication that the Intel RNG performance is within the guidelines specified by FIPS 140-1.

On system startup, the FIPS 140-1 statistical RNG tests are performed on the Intel RNG. An application can also perform these tests via the CSSM passthrough function. Passthrough functions are used to provide additional functions not available through the standard CSSM APIs. Since the CSSM API does not provide a specific RNG test API, CSPs supporting the Intel RNG provide this feature via a passthrough function. Example 4 shows how applications make the passthrough function call for the FIPS 140-1 statistical RNG tests.

```
// Example of how to test the Intel RNG via CSSM

CSSM_CSP_HANDLE hCSP;
CSSM_CONTEXT testCxt;
CSSM_CC_HANDLE hCC;
CSSM_BOOL *passTest;

// Assume a CSP supporting the Intel RNG, hCSP, was already found and attached
// Create a context for the call to the passthrough function
hCC = CSSM_CSP_CreatePassThroughContext( hCSP, NULL, NULL, 0 );

//Create a CSSM_CONTEXT with parameters for the passthrough function
testCxt.ContextType = CSSM_ALGCLASS_RANDOMGEN;
testCxt.AlgorithmType = CSSM_ALGID_IntelPlatformRandom;
testCxt.Reserve = 0;
testCxt.NumberOfAttributes = 0;
testCxt.ContextAttributes = 0;

// Make the call to test the RNG using the crypto context and CSSM_CONTEXT
//previously created.
passTest = CSSM_CSP_PassThrough( hCC, CSP_PASSTHROUGH_TEST_RNG, (void *)(&testCxt));

// check the return value to determine if the test passed.
if (*passTest == CSSM_FALSE )
   // The test failed for some reason
else
   // Test passed
```

Example 4: Performing the FIPS 140-1 tests on the Intel RNG via CSSM API calls

In this example, an application creates a security context for the passthrough function. The created security context is used to specify the CSP. Next, the application creates the parameters for the passthrough function and stores them in a `CSSM_CONTEXT` structure. The application calls **`CSSM_CSP_PassThrough`**, using the `CSP_PASSTHROUGH_TEST_RNG` passthrough ID. A CSP uses the passthrough to determine the appropriate passthrough function to execute. The FIPS 140-1 RNG test function returns `CSSM_TRUE` if all RNG tests successfully pass based on the criteria specified in the FIPS140-1 publication. If any of the tests fail, the passthrough function returns `CSSM_FALSE`.

ISVs should remember that the FIPS140-1 tests require 20,000 random bits. For applications with time-sensitive constraints, care should be taken concerning when, or if, the FIPS 140-1 tests should be performed.

Conclusion

Random numbers are the foundation of secure cryptographic solutions, digital signatures, and protected communications protocols. The Intel RNG provides a high quality, hardware-based random number generator suitable as a fundamental security building block. Existing CDSA-based applications can easily upgrade current random number usage to the Intel RNG with minor changes to the application's existing code base. Easy access to the Intel RNG through the CDSA security framework protects existing CDSA development efforts by ISVs.

References

[1] Davies, Robert. "True Random Numbers." http://webnz.com/robert/true_rng.html (9 Oct. 1998).

[2] "Diehard." http://stat.fsu.edu/~geo/diehard.html (16 Oct. 1998)

[3] Ellison, Carl. "Cryptographic Random Numbers." Draft P1363 Appendix E. http://www.clark.net/pub/cme/P1363/ranno.html (9 Oct. 1998).

[4] Schneier, Bruce. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. New York: John Wiley & Sons, 1996.

[5] FIPS 140-1, "Security Requirements for Cryptographic Modules." Federal Information Processing Standards Publication 140-1. U.S. Department of Commerce/NIST, National Technical Information Service. Springfield, Virginia, 1994. http://csrc.ncsl.nist.gov/fips/fips1401.htm (16 Oct. 1998)

[6] The Open Group, "Common Security: CDSA and CSSM". http://www.opengroup.org/pubs/catalog/c707.htm (Dec. 1997)